RV'12, LNCS. to appear

Maximal Causal Models for Sequentially Consistent Systems*

Traian Florin Șerbănuță[1,2], Feng Chen[1], and Grigore Roșu[1,2]

[1] University of Illinois at Urbana-Champaign
[2] University "Alexandru Ioan Cuza" Iași

Abstract. This paper shows that it is possible to build a *maximal and sound* causal model for concurrent computations from a given execution trace. It is sound, in the sense that any program which can generate a trace can also generate all traces in its causal model. It is maximal (among sound models), in the sense that by extending the causal model of an observed trace with a new trace, the model becomes unsound: there exists a program generating the original trace which cannot generate the newly introduced trace. Thus, the maximal sound model has the property that it comprises *all* traces which *all* programs that can generate the original trace can also generate. The existence of such a model is of great theoretical value as it can be used to prove the soundness of non-maximal, and thus smaller, causal models.

1 Introduction

Traces of events describing concurrent computations have been employed in a plethora of methods for testing and analyzing concurrent systems. The common pattern for all these methods (e.g., [2,3,5,7,12–16,18–21]) is: (1) the program is instrumented to trace the execution of programs; then (2) *one* execution trace is recorded; then (3) an abstraction of that trace, i.e., a *model*, is derived; and finally, (4) the obtained model is used to "predict" (problematic) event patterns occurring in other possible executions abstracted by it.

Consider, for example, the conventional happens-before causality: if two conflicting accesses to an object are not causally ordered, then a data-race is reported [15]. But is this the best one can do? Of course, not. A series of papers propose more relaxed happens-before causal models where one can also permute blocks protected by the same lock, provided that they access disjoint variables [12], thus discovering new concurrency bugs not observable with plain happens-before. But is this the best one can do? Of course, not. Other papers propose models where one can also permute semantic blocks provided that each read access continues to correspond to the same write [16,18,21]. Others go even further. Section 5 discusses a series of existing causal models; we only study *sound* models here, i.e., ones which only report real problems in the analyzed systems, allowing

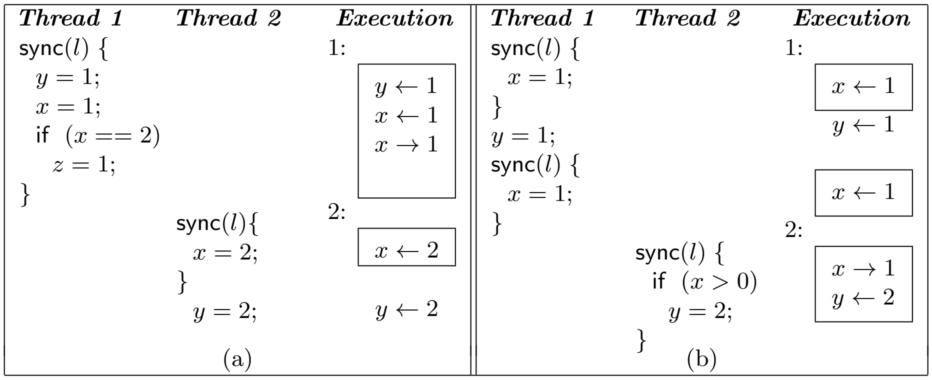
* This work was supported in part by Contract 161/15.06.2010, SMISCSNR 602-12516 (DAK) and by NSF grant CCF-0916893.

developers to focus on fixing those real problems and not on additionally sorting them out from false positives. We would naturally like to know whether there is an end to the question “Is this the best we can do?”, that is, whether there is any causal model that can be associated to a given execution trace which comprises the maximum number of causally equivalent traces.

Although most runtime analysis techniques are built upon some underlying sound causal model, possibly relaxed for efficiency reasons, each effort seems to focus more on how to capture it efficiently rather than proving its soundness (often implicitly assumed) or studying its relationship to existing models (other than empirically comparing the number of found bugs). Moreover, since such approaches attempt to extract information from *one* observed trace and to find property violations, they actually deal with *causal properties* (e.g., causal datarace, causal atomicity), which are instances of desired system-wise properties that can be detected using only the causal information gathered from the observed trace. Since what can be inferred from a trace intrinsically depends on the chosen causal model, definitions of causal properties differ from technique to technique, with the undesirable effect that a causal property (e.g., a datarace) in one model might not be recognized as such by another model.

1.1 Motivating Examples

Each example in Figure 1 shows a two-threaded program, together with one of its possible executions, in which Thread 1 is executed completely before Thread 2 starts. In this representation of executions, synchronized blocks are boxed, while write and read operations on shared locations are denoted by ← (receiving a value), and → (yielding a value), respectively. Both *programs* exhibit a race condition between the two write operations on y. However, are the observed executions also exhibiting a causal datarace?



```
Thread 1        Thread 2       Execution
sync(l) {                      1:
  y = 1;                         y ← 1
  x = 1;                         x ← 1
  if (x == 2)                    x → 1
    z = 1;
}
                sync(l){       2:
                  x = 2;         x ← 2
                }
                  y = 2;         y ← 2
                     (a)
```

```
Thread 1        Thread 2       Execution
sync(l) {                      1:
  x = 1;                         x ← 1
}                                y ← 1
y = 1;
sync(l) {                        x ← 1
  x = 1;
}                              2:
                sync(l) {        x → 1
                  if (x > 0)     y ← 2
                    y = 2;
                }
                     (b)
```

Fig. 1. Motivating examples.

When analyzing the observed execution in Figure 1(a), a simple happens-before approach ordering all accesses to concurrent objects [15] cannot observe a causal datarace: the release operation of the lock in Thread 1 is required to happen-before the acquire of the lock in Thread 2. Happens-before with lock atomicity [12] is not able to infer a causal datarace either: although the lock atomicity would allow for the two lock-blocks to be permuted, the read of x in Thread 1 is still required to happen-before the write of x in Thread 2. Yet, the race condition can be captured as a causal datarace of the observed execution by weaker happens-before models [16, 18, 21], since in those models, one can additionally permute a write before a read of same location, as long as it is permuted before the write corresponding to that read. Thus, the trace generated by the program in Figure 1(a) *has or does not have* a causal datarace, depending upon the particular causal model employed.

However, none of the approaches mentioned above can detect the race condition in Figure 1(b) as a causal datarace for the observed execution. The reason for this is that all models enforce at least the read-after-write dependency (i.e., a read should always follow the latest write event of the same variable), and therefore would not allow the permutation of the last two lock-blocks of the execution, since the read of x in Thread 2 must follow the last write of x in Thread 1. Nevertheless, there is enough information in the observed execution to be able to detect the race: since both writes of x in Thread 1 write the same value, it is actually possible to permute the last two lock blocks, and thus detect the race. Moreover, since one could conceive a technique specialized for finding such cases, it can be rightfully claimed that the observed execution has in fact a causal datarace, although not captured by any existing definition.

Given this ever increasing (regarding coverage) sequence of causal models and definitions for causal properties, it is only natural to ask the following question:

> *Is there any causal model that generalizes all existing models, and which cannot be surpassed?*

We answer this question positively in the context of sequential consistency [9]. While we believe the presented approach can be applied to other memory models, we chose sequential consistency here for two reasons: (1) it is broadly accepted, popular and intuitive; (2) it is subsumed by other memory models: errors detected under sequential consistency are also errors for other memory models.

Contributions. The main result of this paper is a semantic framework that allows to prove maximality of causal models, and a proof that our proposed model is indeed the maximal causal model for the observed execution. This means that it comprises precisely *all* traces which can be generated by all programs which can generate the observed trace. Concretely, we show that: (1) all programs which can produce the observed execution can generate all traces in the model; and (2) for any trace not in the model there exists a program generating the observed trace which cannot generate it. To our knowledge, this is the first such result for causal models. We then prove (the implicitly assumed) soundness for a series of existing causal models by showing they are submodels of the proposed model.

Paper structure. Section 2 introduces some notation and discusses sequential consistency. Section 3 axiomatizes consistent concurrent systems and defines our proposed causal models. Section 4 formally defines the maximality claim and proves our model maximal among sound models. Section 5 shows how existing models are included in ours, thus proving their soundness. Section 6 reviews related research and discusses several research ideas connected with the presented work. Section 7 concludes.

2 Execution Model

Assume a machine that can execute arbitrarily many threads in parallel. The execution environment contains a set of *concurrent objects* (shared memory locations, locks, . . .), which are accessed by threads to share data and synchronize. *Threads*, which can only interact through the execution environment, are abstracted as *sequences of operations on concurrent objects*. The only source of thread non-determinism is the execution environment, that is, if the interaction between a thread and the environment is the same across executions, the thread will execute the same operations, in the same order. To simplify the presentation, we assume no dynamic creation of threads (this presents no technical difficulty).

2.1 Concurrent Objects, Serial Specification

We adopt the definition of concurrent objects and serial specifications proposed by Herlihy and Wing [8]. A concurrent object is behaviorally defined through a set of atomic operations, which any thread can perform on it, and a serial specification of its legal behavior in isolation. The serial specification describes the valid sequences of operations which can be performed on the object. We next describe two common types of concurrent objects.

Shared memory locations Each shared memory location can be regarded as a shared object with read and write operations, whose serial specification states that each read yields the same value as the one of the previous write. Moreover, to avoid non-determinism due to the initial state of the memory, we will further require that all memory locations are initialized, that is, the first operation for each location is a write.

Mutexes Each mutex can be regarded as a concurrent object providing *acquire* and *release* operations. Their mutual exclusion property is achieved through the serial specification which accepts only those sequences in which the difference between the number of *acquire* and *release* operations is either 0 or 1 for each prefix, and all consecutive pairs of *acquire*-*release* share the same thread.

To keep the proofs simple and the concepts clear, we refrain here from adding more concurrency constructs (such as spawn/join, wait/notify, or semaphores). Note, however, that this would not introduce additional complexity, but just further constrain the notion of consistency.

2.2 Events and Traces

Operations performed by threads on concurrent objects are recorded as *events*. We consider events to be abstract entities from an infinite "collection" *Events*, and describe them as tuples of *attribute-value* pairs. The only attributes considered here are: *thread*—the unique id of the thread generating the event, *op*—the operation performed (e.g., *write*, *read*, *acquire*, or *release*), *target*—the concurrent object accessed by the event, and *data*—the value sent/received by the current event, if such exists (e.g., for the *write*/*read* operations). For example, ($\mathit{thread}{=}t_1$, $\mathit{op}{=}\mathit{write}$, $\mathit{target}{=}x$, $\mathit{data}{=}1$) describes an event recording a write operation by thread t_1 to memory location x with value 1. When there is no confusion, we only list the attribute values in an event, e.g., $(t_1, \mathit{write}, x, 1)$. Our choice for deciding what attributes to record in an event considers a monitor which can observe memory and synchronization operations and the identity of the thread performing them, but has no access to the actual code. Section 6 includes a discussion on possible variations on the set of attributes recorded for an event.

For any event e and attribute *attr*, $\mathit{attr}(e)$ denotes the value corresponding to the attribute *attr* in e, and $e[v/\mathit{attr}]$ to denote the event obtained from e by replacing the value of attribute *attr* by v. An *execution trace* is abstracted as a sequence of events. Given a trace τ, a concurrent object o and a thread t, let $\tau\restriction_o$ and $\tau\restriction_t$ denote the restriction of τ to events involving only o, and only t, respectively. Let $\mathit{latest}_o(\tau)$ be the latest event of τ having the *op* attribute o. If o is omitted, it simply means the latest event in τ.

Sequential consistency can be now elegantly defined:

Definition 1 ([1]). *Let τ be any trace.*
(1) τ is ***legal*** *if and only if $\tau\restriction_o$ satisfies o's serial specification for any object o;*
(2) An ***interleaving*** *of τ is a trace τ' such that $\tau'\restriction_t = \tau\restriction_t$ for each thread t.*
(3) A trace τ is ***(sequentially) consistent*** *if it admits a legal interleaving.*

Since we restrict ourselves to sequential consistency, from here on when we say that a trace is sequentially consistent we automatically mean that it is also legal.

3 Feasibility Model

This section introduces an axiomatization for a machine producing consistent traces, and uses it to associate a sound-by-definition causal model to any observed execution, comprising all executions which can potentially be inferred from that execution alone, without additional knowledge of the system generating it.

The two properties/axioms (presented below) we base our approach on are *trace consistency* and *feasible executions*. A consistent trace (Definition 1) disallows "wrong" behaviors, such as reading a value different from the one which was written, or proceeding when a lock cannot be acquired. Feasible executions, defined below, refer to *sets* of execution traces and aim at capturing *all* the behaviors that a given system or program can manifest. No matter what task a concurrent system or program accomplishes, its set of traces must obey some basic properties. First, feasible traces are generate-able, meaning that any prefix of any feasible trace is also feasible; this is captured by our first axiom of feasible traces, *prefix closedness*. Second, we assume that thread interleaving is the only source of non-determinism in producing traces; this is captured by our second axiom of feasible traces, *local determinism*.

Each particular multithreaded system or programming environment, say $\mathcal{S}$, has its own notion of feasible execution, given by its specific intended semantics. Let us call all (possibly incomplete) traces that $\mathcal{S}$ can yield $\mathcal{S}$*-feasible*, and let *feasible*$(\mathcal{S})$ be their set. Instead of defining *feasible*$(\mathcal{S})$, which requires a formal definition of $\mathcal{S}$ and is therefore $\mathcal{S}$-specific (and tedious), we here *axiomatize* it:

Prefix Closedness *Events are indivisible and generated in execution order; hence, feasible*$(\mathcal{S})$ *must be prefix closed*: if $\tau_1\tau_2$ is $\mathcal{S}$-feasible, then τ_1 is $\mathcal{S}$-feasible. Prefix closedness ensures that each event is generated individually, with the possibility of interleaving happening in-between any of them. For example, although the `++ x` instruction generates two events, a read, follow by a write on `x` these are not necessarily consecutive: if the instruction is not properly synchronized, another thread could write `x` after the first event, yielding an atomicity violation.

Local Determinism *The execution of a concurrent operation is determined by the previous events in the same thread, and can happen at any consistent moment after them.* Formally, if $\tau e, \tau' \in \mathit{feasible}(\mathcal{S})$ and $\tau\lceil_{\mathit{thread}(e)} = \tau'\lceil_{\mathit{thread}(e)}$ then: if $\tau' e$ is consistent then $\tau' e \in \mathit{feasible}(\mathcal{S})$; moreover, if $\mathit{op}(e) = \mathit{read}$ and there exists an event e' such that $e = e'[\mathit{data}(e)/\mathit{data}]$ and $\tau' e'$ is consistent, then $\tau' e \in \mathit{feasible}(\mathcal{S})$. The second part says that if a *read* operation is enabled, i.e., all previous events have been generated, then it can be executed at any consistent time (despite the fact that the value it receives might be different from that observed in the original trace). Allowing traces where read events observe a different value than in the original trace might seem like a source of unsoundness. Note, however, that the same local determinism property prohibits the thread on which such a read event occurred to continue after producing this event, by stating that an additional event for a thread is generated *only* if the current trace for that thread is *exactly* the same (including the value) as in the original trace. Suppose, for example, that two threads, identified by t_1 and t_2, assign 1, then execute an increment operation on the same location l. One potential observed trace could be: $(t_1, \mathit{write}, l, 1)(t_2, \mathit{write}, l, 1)(t_1, \mathit{read}, l, 1)(t_1, \mathit{write}, l, 2)(t_2, \mathit{read}, l, 2)(t_2, \mathit{write}, l, 3)$. Local determinism ensures that we can also obtain the (partial) trace

$$(t_1, \mathit{write}, l, 1)(t_2, \mathit{write}, l, 1)(t_1, \mathit{read}, l, 1)(t_2, \mathit{read}, l, 2)(t_1, \mathit{write}, l, 2).$$

This shows that we can use local determinism to interleave threads differently than their original scheduling, as long as consistency is respected and threads produce the same events. Note that, (1) event $e = (t_2, \mathit{read}, l, 2)$ can be generated although it reads a different value than it originally did; and (2) thread t_1 can continue after e was generated (since it concerns a different thread), but thread t_2 cannot (because, e.g., e could be guarding a control statement).

Definition 2. *$\mathcal{S}$ is* ***consistent*** *iff feasible($\mathcal{S}$) satisfies the axioms above.*

A major goal of trace-based analysis is to infer/analyze as many traces as possible using a recorded trace. When one does not know (or does not want to use) the source code of the multithreaded program being executed, one can only infer potential traces of the system resembling the observed trace. Let us now define the proposed causal model, termed *feasibility closure*, as the set of executions which can be inferred from an observed execution—they correspond to the traces obtainable from τ using the feasibility axioms.

Definition 3. *The* ***feasibility closure*** *of a consistent trace τ, written feasible(τ), is the smallest set of traces containing τ which is prefix-closed and satisfies the local determinism property. A trace in feasible(τ) is called* ***τ-feasible****.*

Without dwelling into details here, as this is proved elsewhere [17], intuitively the feasibility closure of a trace contains all interleavings of the observed trace, where each thread is stopped once it read a value from the memory different from the one observed originally, as well as all prefixes of these traces.

The following result formalizes the soundness of the proposed model. Assuming the base axioms are sound, the closure properties guarantee that all traces in our causal model are feasible. In addition, Proposition 1 shows that *any* system/program which can generate one trace, can also generate *all* traces comprised by its causal model.

Proposition 1. *If $\mathcal{S}$ consistent and $\tau \in$ feasible($\mathcal{S}$) then feasible(τ) $\subseteq$ feasible($\mathcal{S}$). Moreover, if τ' is consistent and $\tau \in$ feasible(τ'), then feasible(τ) $\subseteq$ feasible(τ').*

The intuition for $\tau \in$ *feasible*(τ') is that if a run of any program executed on $\mathcal{S}$ can produce τ', then there is also some run of the same program executed also on $\mathcal{S}$ that can produce τ.

4 Maximality

In this section we show that the proposed causal model is *maximal* among sound models, in the sense that any extension to it is done at the expense of soundness. We will prove therefore that given a trace τ' which is not in the feasibility closure of a trace τ, there exists a program p which can generate τ but not τ'; therefore, if the model were extended to include τ' and used τ' as a witness that a property is satisfied/invalidated by a program generating τ, this would be a false witness if the program which generated τ was p.

To prove our claim, we propose CONC, a very simple (not even Turing complete) concurrent language. The benefit of such a simple language is that it can conceivably be simulated in any real language; therefore, proving the maximality result for CONC proves the model is maximal for all languages. Figure 2 presents the grammar and SOS semantics of CONC. The grammar specifies a parallel composition of named threads. Each thread is a succession of statements and uses one internal register to load data from the shared memory.

CONC SYNTAX:

$$\begin{aligned} \mathit{Proc} &::= \mathit{Proc} \mathbin{||} \mathit{Proc} \mid \mathit{Int} : \mathit{Stmt} \\ \mathit{Stmt} &::= \mathit{Stmt}\ ;\ \mathit{Stmt} \mid \mathsf{nop} \mid \mathsf{if}\ \mathit{Int}\ \mathsf{then}\ \mathit{Stmt} \\ &\quad \mid \mathsf{load}\ \mathit{Loc} \mid \mathit{Loc} := \mathit{Int} \mid \mathsf{acquire}\ \mathit{Loc} \mid \mathsf{release}\ \mathit{Loc} \end{aligned}$$

CONC SEMANTICS:

$$\frac{\langle p_1, \sigma, \delta, \rho\rangle \xrightarrow{\tau} \langle p_1', \sigma', \delta', \rho'\rangle}{\langle p_1 \mathbin{||} p_2, \sigma, \delta, \rho\rangle \xrightarrow{\tau} \langle p_1' \mathbin{||} p_2, \sigma', \delta', \rho'\rangle} \qquad (\mathit{Par}_1)$$

$$\frac{\langle p_2, \sigma, \delta, \rho\rangle \xrightarrow{\tau} \langle p_2', \sigma', \delta', \rho'\rangle}{\langle p_1 \mathbin{||} p_2, \sigma, \delta, \rho\rangle \xrightarrow{\tau} \langle p_1 \mathbin{||} p_2', \sigma', \delta', \rho'\rangle} \qquad (\mathit{Par}_2)$$

$$\frac{\langle s, \sigma, \delta, \rho, t\rangle \xrightarrow{\tau} \langle s', \sigma', \delta', \rho', t\rangle}{\langle t : s, \sigma, \delta, \rho\rangle \xrightarrow{\tau} \langle t : s', \sigma', \delta', \rho'\rangle} \qquad (\mathit{Thread})$$

$$\frac{\langle s_1, \sigma', \delta', \rho', t\rangle \xrightarrow{\tau} \langle s_1', \sigma', \delta', \rho', t\rangle}{\langle s_1\ ;\ s_2, \sigma, \delta, \rho, t\rangle \xrightarrow{\tau} \langle s_1'\ ;\ s_2, \sigma', \delta', \rho', t\rangle} \qquad (\mathit{Seq})$$

$$\frac{\cdot}{\langle \mathsf{nop}\ ;\ s, \sigma, \delta, \rho, t\rangle \xrightarrow{\epsilon} \langle s, \sigma, \delta, \rho, t\rangle} \qquad (\mathit{Nop})$$

$$\frac{\cdot}{\langle \mathsf{if}\ i\ \mathsf{then}\ s, \sigma, \delta, \rho, t\rangle \xrightarrow{\epsilon} \langle s, \sigma, \delta, \rho, t\rangle} \qquad \textbf{if } \rho(t) = i \quad (\mathit{If}_{\mathit{true}})$$

$$\frac{\cdot}{\langle \mathsf{if}\ i\ \mathsf{then}\ s, \sigma, \delta, \rho, t\rangle \xrightarrow{\epsilon} \langle \mathsf{nop}, \sigma, \delta, \rho, t\rangle} \qquad \textbf{if } \rho(t) \neq i \quad (\mathit{If}_{\mathit{false}})$$

$$\frac{\cdot}{\langle \mathsf{load}\ x, \sigma, \delta, \rho, t\rangle \xrightarrow{(t, \mathit{read}, x, i)} \langle \mathsf{nop}, \sigma, \delta, \rho[t \leftarrow i], t\rangle} \qquad (\mathit{Read})$$
$$\textbf{where } i = \sigma(x)$$

$$\frac{\cdot}{\langle x := i, \sigma, \delta, \rho, t\rangle \xrightarrow{(t, \mathit{write}, x, i)} \langle \mathsf{nop}, \sigma[x \leftarrow i], \delta, \rho, t\rangle} \qquad (\mathit{Write})$$

$$\frac{\cdot}{\langle \mathsf{acquire}\ x, \sigma, \delta, \rho, t\rangle \xrightarrow{(t, \mathit{acquire}, x)} \langle \mathsf{nop}, \sigma, \delta[x \leftarrow t], \rho, t\rangle} \qquad (\mathit{Acq})$$
$$\textbf{if } \delta(x) = \perp$$

$$\frac{\cdot}{\langle \mathsf{release}\ x, \sigma, \delta, \rho, t\rangle \xrightarrow{(t, \mathit{release}, x)} \langle \mathsf{nop}, \sigma, \delta[x \leftarrow \perp], \rho, t\rangle} \qquad (\mathit{Rel})$$
$$\textbf{if } \delta(x) = t$$

Fig. 2. Syntax and SOS semantics for the CONC language

load x loads the value at location x into the internal register of the thread, $x := i$ stores integer i at location x, acquire and release have the straight-forward semantics, and if i then s executes s only if the internal register has value i. A running configuration of CONC is a tuple $\langle p, \sigma, \delta, \rho\rangle$ where p is the remainder of the program being executed, σ maps variables to values, δ maps each lock to the id of the thread holding it, and ρ gives for each thread the value of its internal register. In the SOS derivation rules we additionally use configurations of the form $\langle p, \sigma, \delta, \rho, t\rangle$, where t is the thread id obtained in the (Thread) rule, which is propagated by all following rules. Assuming p has n threads, the initial configuration of the system is $\mathit{START}(p) = \langle p, \sigma_\epsilon, \delta_\epsilon, \rho^n_\epsilon \rangle$ where σ_ϵ, δ_ϵ, and ρ^n_ϵ, initialize all locations, locks, and registers for the n threads with $\bot$, respectively.

We have chosen this minimal language both because it is sufficiently expressive to generate all (finite) legal traces, and because it is quite easy to mimic in any other language. In Java, for example, each thread would be modeled by a thread object, and all threads could be started in a loop by the main thread. Since beginnings of threads do not generate events, this is as-if all threads start together in parallel. The running method of each Java thread object would declare a local variable r to stand for the register, and then the two CONC instructions dealing with the register translate as follows: $\mathsf{load}\ l$ becomes $r = l$, and $\mathsf{if}\ \ i\ \ \mathsf{then}\ \ s$ becomes $\mathsf{if}\ \ (r == i)\quad s$.

It is straightforward to associate to each event an instruction producing it. Let *code* be the mapping defined on events as follows:

$$\mathit{code}(e) = \begin{cases} \mathsf{load}\ x & \textbf{if } e = (t, \mathit{read}, x, i) \\ x := i & \textbf{if } e = (t, \mathit{write}, x, i) \\ \mathsf{acquire}\ x & \textbf{if } e = (t, \mathit{acquire}, x) \\ \mathsf{release}\ x & \textbf{if } e = (t, \mathit{release}, x) \end{cases}$$

Given a program p, let $p\restriction_t$ be its projection on thread t, that is, the statement labeled by t in the parallel composition.

The following result shows that, except for the code, the running configuration is completely determined by the trace generated up to that point:

Proposition 2. *If* $\mathrm{CONC} \vdash \mathit{START}(p) \xrightarrow{\tau}{}^{*} \langle p', \sigma_\tau, \delta_\tau, \rho^n_\tau \rangle$, *where* n *is the number of threads of* p, *then:*

(1) $\sigma_\tau(x) = \mathit{data}(\mathit{latest}_{\mathit{write}}(\tau\restriction_x))$*;*
(2) $\delta_\tau(x) = \begin{cases} \mathit{thread}(\mathit{latest}(\tau\restriction_x)),\ \mathit{ifop}(\mathit{latest}(\tau\restriction_x)) = \mathit{acquire} \\ \bot,\ \mathit{otherwise} \end{cases}$*;*
(3) $\rho^n_\tau(t) = \mathit{data}(\mathit{latest}_{\mathit{read}}(\tau\restriction_t))$*.*

Therefore, in the sequel we will use $\mathrm{CONC} \vdash p \xrightarrow{\tau}{}^{*} p'$ instead of $\mathrm{CONC} \vdash \mathit{START}(p) \xrightarrow{\tau}{}^{*} \langle p', \sigma_\tau, \delta_\tau, \rho^n_\tau \rangle$

Now, let us prove that the semantics of CONC does indeed satisfy the sequential consistency axioms. Let p be a CONC program and let *feasible*(p) be the set of all p-feasible traces; that is τ is p-feasible if there exists a program p' such that $\mathrm{CONC} \vdash p \xrightarrow{\tau}{}^{*} p'$. We sill show that *feasible*(p) satisfies the *strong local determinism* property, namely, not only that an enabled event can be generated at any point by its thread, but that it also must be the (unique) next event generated by that thread (ignoring the *data* attribute for *read* events). Formally,

Definition 4. *feasible*(p) *satisfies the* ***strong local determinism*** *property if it satisfies local determinism and if* $\tau_1 e_1$ *and* $\tau_2 e_2$ *are* p*-feasible,* $\mathit{thread}(e_1) = \mathit{thread}(e_2) = t$*, and* $\tau_1\restriction_t = \tau_2\restriction_t$*, then* $\mathit{op}(e_1) = \mathit{op}(e_2)$*, and* $\mathit{target}(e_1) = \mathit{target}(e_2)$*; if additionally* $\mathit{op}(e_i) = \mathit{write}$*, then also* $\mathit{data}(e_1) = \mathit{data}(e_2)$*.*

The following result shows that every CONC program p is a consistent system in the sense of Definition 2.

Proposition 3 (CONC consistency). *$feasible(p)$ satisfies prefix closedness and strong local determinism.*

Proof (Sketch). Prefix closedness is obvious, since the semantics can emit at most one event for each execution step. The second property follows by case analysis on the rule SOS rule applied to produce the relevant event for local determinism.

Now, given a trace τ, let us build the canonical CONC program generating it. *code* can be naturally extended on traces by $code(e\tau) = code(e)\ ;\ code(\tau)$. Let $\{t_1, t_2, \ldots, t_n\}$ be the set of thread ids appearing in τ. Then the program associated to a trace τ is defined by $program(\tau) = t_1 : code(\tau{\restriction_{t_1}}) \mid\mid \cdots \mid\mid t_n : code(\tau{\restriction_{t_n}})$.

Let us also define the empty program with n threads as $program^n(\epsilon) = t_1 :$ nop $\mid\mid \cdots \mid\mid t_n :$ nop. The following result shows that the program corresponding to a consistent trace can indeed generate that trace.

Proposition 4. *If τ is a consistent trace with n threads, then* $\mathrm{CONC} \vdash program(\tau) \xrightarrow{\tau}^{*} program^n(\epsilon)$.

The following theorem justifies the maximality claims for the proposed model.

Theorem 1 (Maximality). *For any consistent trace τ' which is not τ-feasible there exists a program generating τ but not τ'.*

Proof (Sketch). Because of prefix closeness and thread determinism, the only interesting case to analyze is when τ' continues the execution on a thread after reading a value distinct from the one recorded in an event e of τ. in that case, we create a new program p from $program(\tau)$ by inserting a conditional write instruction right after that generating event e. We then show that program p can still generate τ, but cannot generate τ'.

5 Proving Soundness for Existing Causal Models

Focusing on identifying concurrency anomalies and measuring success based on the number of bugs found, almost no causal model in the literature is actually proved sound. The authors of a causal model usually give some common-sense arguments for their choice and informally rely on the soundness of Happens-Before [9]. However, intuition can sometimes be misleading: in Section 5.4 we reveal a soundness problem with the model of Sen et al. [16]. Moreover, even when proved sound, the proofs are quite laborious, each having to repeat the formalization of an execution model. Proving soundness of other causal models by embedding them in our already proven sound model eliminates the need for an execution model and reduces proofs to checking closure properties.

We start with the following result, which can be regarded as a sufficient criterion for feasibility:

Theorem 2. *Any consistent prefix of an interleaving of τ is τ-feasible.*

The remainder of this section shows that existing sound causal models are captured by the feasibility closure as simple instances of Theorem 2. Another important consequence of Theorem 2 is that is basically shows there is a unique feasibility closure associated to a concurrent computation, regardless of the representative trace [17].

5.1 Happens Before Relation on Mazurkiewicz Traces

One elegant way to capture the happens-before trace equivalence is the Mazurkiewicz trace [10] associated to the dependence given by the happens-before relation.

The happens-before dependence is a set $T \cup D$, where $T = \bigcup_t \{(e_1, e_2) : \tau\restriction_t = \tau_1 e_1 e_2 \tau_2\}$ is the intra-thread sequential dependence relation and $D = \bigcup_x \{(e_1, e_2) : \tau\restriction_x = \tau_1 e_1 e_2 \tau_2$ *such that* e_1 *or* e_2 *is a write of* $x\}$ is the sequential memory dependence relation. Given this happens-before dependence, the Mazurkiewicz trace associated with τ is defined as the least set $[\tau]$ of traces containing τ and being closed under permutation of consecutive independent events [10]: if $\tau_1 e_1 e_2 \tau_2 \in [\tau]$ and $(e_1, e_2) \notin T \cup D$, then $\tau_1 e_2 e_1 \tau_2 \in [\tau]$.

The following result shows that the feasibility closure is closed under the equivalence relation generated by happens-before, that is, happens-before is captured by our model, and thus re-shown sound for consistent executions:

Proposition 5. *If $\tau_1 e_1 e_2 \tau_2$ is τ-feasible and $(e_1, e_2) \notin T \cup D$, then $\tau_1 e_2 e_1 \tau_2$ is τ-feasible. Given any τ-feasible trace τ', $[\tau'] \subseteq feasible(\tau)$. Hence, $[\tau] \subseteq feasible(\tau)$.*

5.2 Weak Happens Before

Several more recent trace analysis techniques [16, 18, 21] argue that the happens-before model can be further relaxed, noticing that the only purpose of the write-after-read happens-before order is to guarantee that a read event always reads the same write event as before in any feasible interleaving of the original trace. Therefore, one only needs to preserve the *read-after-write dependence*:

Definition 5. *Suppose $\tau = \tau_1 e_1 \tau_2 e_2 \tau_3$. Then e_2 **write-read depends on** e_1 in τ, written $e_1 <^{wr}_{\tau} e_2$, if $target(e_1) = target(e_2)$, $op(e_1) = write$, $op(e_2) = read$, and for all $e \in \mathcal{E}_{\tau_2}$, either $target(e) \neq target(e_1)$, or $op(e) \neq write$.*

That is, $e_1 <^{wr}_{\tau} e_2$ iff the value read by e_2 is the value written by e_1.

Sen et al. [16] introduce the notion of *atomic* sets associated to each *write* event, containing itself and all read events which write-read depend on it, accepting as feasible executions all linearizations of the transitive closure of the combined $<^{wr}_{\tau}$ and thread ordering, satisfying the additional requirement that the atomic sets are preserved. However this can be simply restated as follows [21]:

Definition 6. *$\tau \sim \tau'$ if τ is an interleaving of τ' and $<^{wr}_{\tau} = <^{wr}_{\tau'}$.*

That is, the $\sim$-equivalence class of τ contains all interleavings of τ which have exactly the same write-read dependence relation. Next result shows that this model is also captured by our model.

Proposition 6. *If τ_1 is τ-feasible, and $\tau_1 \sim \tau_2$, then τ_2 is also τ-feasible.*

5.3 Happens-Before with synchronization

A conservative and sound approach, requiring no implementation changes, to handle locks in happens-before-based trace analysis techniques is to assume that *acquire* and *release* operations on the same lock yield the same happens-before dependence as if they were particular *write* and *read* operations (on the lock variable) [15]. However, this prevents synchronized blocks from being permuted, and thus imposes coverage limitations. The lock-set approaches, also called hybrid happens-before [12], propose to handle locks separately, associating with each event the set of locks [14] protecting them, hereby not enforcing any particular order between synchronized blocks.

We here group the events protected by locks in *atomic blocks*. Events e_1 and e_2 from a consistent trace τ, both generated by thread t, are *l-atomic* in τ, written $e_1 \Updownarrow_l^\tau e_2$, if and only if there is some *acquire* event e on lock l generated by t before both e_1 and e_2, and there is no *release* event e' on l generated by t between e and either of e_1, e_2. For each lock l, let $[e]_l$ denote the *l-atomic equivalence class of e*. Assuming a trace in which all acquired locks are eventually released, l-atomic equivalence classes consist of all events belonging the the same acquire-release block of l. A trace τ' is *consistent with the lock atomicity of τ* if there exists no lock l and decomposition $\tau_1 e_1 \tau_2 e_2 \tau_3 e_3 \tau_4 e_4 \tau_5$ such that $e_1 \Updownarrow_l^\tau e_3$ and $e_2 \Updownarrow_l^\tau e_4$ and $[e_1]_l \neq [e_2]_l$. Let $\prec_{hb}^\tau$ be the transitive closure of the union between happens-before and thread orderings of τ. The following holds:

Proposition 7. *Let τ' be a τ-feasible trace. Any linearization of $\prec_{hb}^{\tau'}$ consistent with the lock atomicity of τ' is τ-feasible.*

5.4 Weak-Happens-Before with synchronization

We next present two approaches to handling synchronization in weak-happens-before models and show they are both embeddable in our model.

Lock atomicity via write-read atomicity [16]. Since the notion of write-read atomicity already allows atomic sets to be permuted, it seems reasonable to use the conservative idea from standard happens-before methods, and treat *acquire* as a *write* event and *release* as a read event. Formally, given the consistent trace τ, one could additionally introduce an atomic dependence relation $<_\tau^a$ given by $e_1 <_\tau^a e_2$ if $\tau = \tau_1 e_2 \tau_2 e_2 \tau_3$, $\mathit{target}(e_1) = \mathit{target}(e_2)$, $\mathit{op}(e_1) = \mathit{acquire}$, $\mathit{op}(e_2) = \mathit{release}$, and there is no event e in τ_2 such that $\mathit{target}(e) = \mathit{target}(e_1)$, and $\mathit{op}(e) = \mathit{acquire}$. With this definition, equivalent traces to an observed trace τ are those interleavings of τ having the same write-read and atomic dependencies.

However, this definition needs a careful approach. Consider the example in Figure 1(b), and suppose that we observe a similar execution, but that the program is stopped after the *read* of x in Thread 2. Since no *release* event has been generated, the *acquire* in Thread 2 has no event depending on it, and thus it can be permuted (without the *read* event on x it was supposed to protect) before the last lock-block of Thread 1. Then, the final *read* of x itself can be permuted past the final *release* of l in Thread 1, exhibiting a spurious causal datarace.

Nevertheless, these models are sound for *synchronization complete* traces, that is, traces in which each acquired lock is eventually released.

Proposition 8. *Let τ_1 be a synchronization complete τ-feasible trace. Any interleaving τ_2 of τ_1 satisfying that $<^{wr}_{\tau_2} = <^{wr}_{\tau_1}$ and $<^{a}_{\tau_2} = <^{a}_{\tau_1}$ is τ-feasible.*

Lock atomicity via locksets. Wang and Stoller [21] propose a weak-happens-before model based on write-read dependence, while using locksets to handle locks as individual objects. In this model, a trace τ' is equivalent with a consistent trace τ if τ' is an interleaving of τ having the same write-read dependence relation and being consistent with the lock atomicity of τ.

Proposition 9. *Let τ_1 be a τ-feasible trace. Any interleaving τ_2 of τ_1, consistent with the lock atomicity of τ_1 and satisfying that $<^{wr}_{\tau_2} = <^{wr}_{\tau_1}$ is τ-feasible.*

6 Related Work and Discussion

Beginning with the introduction of the Happens-Before ordering by Lamport [9], there has been a considerable amount of research on models and techniques to abstract executions for the purpose of inferring causally equivalent executions satisfying/violating particular but important properties, such as dataraces or atomicity/serializability [2, 5, 7, 12, 14–16, 19–21]. Section 5 shows that the sound causal models upon which the above mentioned techniques were based [10, 12, 15, 16, 21] are subsumed by the maximal causal model; their soundness follows as a corollaries of Theorem 1.

Ganai and Gupta [6] apply a similar technique for software model checking, attempting to reduce the state space to be explored using sequential consistency constraints. Similarly, building on a previous draft of this paper, Said et al. [13] encode the axioms of our proposed model (extended with constructs for thread creation and wait/notify) into an SMT solver and use that to effectively search the model for potential dataraces in Java programs.

Another interesting and productive line of research attempts to use information about the actual program code to either statically detect potential bad behaviors [11], or to use information about the program and about the property to be checked to further relax the models of executions [3, 20].

Adding or removing attributes from events. Our choice of which attributes to be included in an event was based on the idea of observing the execution of any multithreaded program executed on any machine offering no guarantees other than sequential consistency. Therefore, no semantical information is assumed about the program other than the identity of the thread performing an operation. There are other possible choices, each with their benefits. For example, Sinha et al. [18] choose not to record the value read/written in the memory. Thus, their model must preserve the read-after-write dependence, and the set of comprised traces is thus comparable with that of Wand and Stoller [21], and Sen et al. [16]. We believe a similar maximality result could be proved for traces of this kind, but that has not been attempted yet. In contrast, Wang et al. [20] enrich the events with symbolic information extracted from the program executing them. This allows them to obtain more comprehensive models at the expense of having to analyze the code. Since analyzing the code statically leads quickly to undecidability issues, and thus static analyzers need to be conservative, we believe there might indeed be no similar maximality result for these types of models, their coverage increasing with the power of the analysis.

Causal properties of traces. Since our model associates for a trace all traces which can be obtained by all programs which can obtain that trace, this allows for program-independent definitions of causal properties. For example,Wang and Stoller [21] propose serializability of a trace τ as the property that there exists an alternative execution of the program producing an interleaving of τ in which each transaction is a sequential block. Farzan and Madhusudan [4] relax this constraint by requiring that for each transaction there exists an alternative execution of the program producing an interleaving of τ containing that transaction as a sequential block. Sen et al. [16] say that a trace exhibits a datarace if there exists an alternative execution of the program producing an (partial) interleaving of τ in which the conflicting events are consecutive.

The program-independent properties associated to any of the above (program-dependent) definitions can be obtained by simply replacing the (rather informal) "alternative execution of the program producing an interleaving of τ" with "a τ-feasible trace", as defined by Definition 3. Formal definitions of these causal properties can be found in the companion technical report [17].

7 Conclusion

We have shown that, by axiomatizing basic properties of (sequentially consistent) concurrent systems, one can obtain *maximally sound causal models* for concurrent executions, which can be naturally associated to each observed trace, capturing all feasible traces which could be inferred from it. The maximality result has two important theoretical implications. First, verifying the soundness claims for any causal model is reduced to proving that it is a submodel of the maximal one. Second, since the maximal model captures all causally equivalent traces, it allows for universal, program-independent definitions for causal properties. Although this paper focuses on proving the maximality claim of our model, the companion technical report [17] additionally provides a constructive characterization of the proposed model, as well as a model checking algorithm.

References

1. H. Attiya and J. L. Welch. Sequential consistency versus linearizability. *TOCS*, 12:91–122, May 1994.
2. U. Banerjee, B. Bliss, Z. Ma, and P. Petersen. A theory of data race detection. In *PADTAD'06*, pages 69–78, New York, 2006. ACM.
3. F. Chen and G. Roșu. Parametric and sliced causality. In *CAV'07*, volume 4590 of *LNCS*, pages 240–253.
4. A. Farzan and P. Madhusudan. Causal atomicity. In *CAV'06*, volume 4144 of *LNCS*, pages 315–328, 2006.
5. C. Flanagan and S. N. Freund. Atomizer: a dynamic atomicity checker for multithreaded programs. In *POPL'04*, pages 256–267, 2004.
6. M. K. Ganai and A. Gupta. Efficient modeling of concurrent systems in BMC. In *SPIN'08*, volume 5156 of *LNCS*, pages 114–133, 2008.
7. D. P. Helmbold, C. E. McDowell, and J. Z. Wang. Determining possible event orders by analyzing sequential traces. *TPDS*, 4(7):827–840, 1993.
8. M. P. Herlihy and J. M. Wing. Linearizability: a correctness condition for concurrent objects. *TOPLAS*, 12:463–492, July 1990.
9. L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess progranm. *IEEE Trans. Comput.*, 28(9):690–691, 1979.
10. A. Mazurkiewicz. Trace theory. In *Advances in Petri nets*, pages 279–324, New York, NY, USA, 1987. Springer-Verlag.
11. M. Naik, A. Aiken, and J. Whaley. Effective static race detection for Java. In *PLDI'06*, pages 308–319, 2006.
12. R. O'Callahan and J.-D. Choi. Hybrid dynamic data race detection. *SIGPLAN Not.*, 38(10):167–178, 2003.
13. M. Said, C. Wang, Z. Yang, and K. A. Sakallah. Generating data race witnesses by an SMT-based analysis. In *NASA Formal Methods'11*, volume 6617 of *LNCS*, pages 313–327, 2011.
14. S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: a dynamic data race detector for multithreaded programs. *TOCS*, 15(4):391–411, 1997.
15. E. Schonberg. On-the-fly detection of access anomalies. *Best of PLDI 1979-1999*, 39:313–327, April 2004.
16. K. Sen, G. Roșu, and G. Agha. Detecting errors in multithreaded programs by generalized predictive analysis of executions. In *FMOODS'05*, volume 3535 of *LNCS*, pages 211–226, 2005.
17. T. F. Șerbănuță, F. Chen, and G. Roșu. Maximal causal models for sequentially consistent systems. Technical Report `http://hdl.handle.net/2142/27708`, University of Illinois at Urbana-Champaign, October 2011.
18. A. Sinha, S. Malik, C. Wang, and A. Gupta. Predictive analysis for detecting serializability violations through trace segmentation. In *MEMOCODE'11*.
19. M. Vaziri, F. Tip, and J. Dolby. Associating synchronization constraints with data in an object-oriented language. In *POPL'06*, pages 334–345, 2006.
20. C. Wang, S. Kundu, M. Ganai, and A. Gupta. Symbolic predictive analysis for concurrent programs. In *FM'09*, pages 256–272, 2009.
21. L. Wang and S. D. Stoller. Accurate and efficient runtime detection of atomicity errors in concurrent programs. In *PPoPP'06*, pages 137–146, 2006.